Durham University

Durham Research Online

Deposited in DRO:

06 October 2014

Version of attached file:

Submitted Version

Peer-review status of attached file:

Not peer-reviewed

Citation for published item:

Weinzierl, Tobias and Bader, Michael and Unterweger, Kristof and Wittmann, Roland (2014) 'Block fusion on dynamically adaptive spacetree grids for shallow water waves.', Parallel processing letters., 24 (3). p. 1441006.

Further information on publisher's website:

http://dx.doi.org/10.1142/S0129626414410060

Publisher's copyright statement:

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014, 1441006, 10.1142/S0129626414410060 © World Scientific Publishing Company http://www.worldscientific.com/worldscinet/ppl

Additional information:

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in DRO
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full DRO policy for further details.

Durham University Library, Stockton Road, Durham DH1 3LY, United Kingdom
Tel : +44 (0)191 334 3042 | Fax : +44 (0)191 334 2971
https://dro.dur.ac.uk

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

BLOCK FUSION ON DYNAMICALLY ADAPTIVE SPACETREE GRIDS FOR SHALLOW WATER WAVES

TOBIAS WEINZIERL

School of Engineering and Computing Sciences, Durham University, Stockton Road Durham DH13LE, Great Britain

and

MICHAEL BADER

KRISTOF UNTERWEGER

ROLAND WITTMANN

Institut für Informatik, Technische Universität München, Boltzmannstr. 3 85748 Garching, Germany

Received April 2014
Revised Juli 2014

Communicated by Guest Editors

ABSTRACT

Spacetrees are a popular formalism to describe dynamically adaptive Cartesian grids. Even though they directly yield a mesh, it is often computationally reasonable to embed regular Cartesian blocks into their leaves. This promotes stencils working on homogeneous data chunks. The choice of a proper block size is sensitive. While large block sizes foster loop parallelism and vectorisation, they restrict the adaptivity's granularity and hence increase the memory footprint and lower the numerical accuracy per byte. In the present paper, we therefore use a multiscale spacetree-block coupling admitting blocks on all spacetree nodes. We propose to find sets of blocks on the finest scale throughout the simulation and to replace them by fused big blocks. Such a replacement strategy can pick up hardware characteristics, i.e. which block size yields the highest throughput, while the dynamic adaptivity of the fine grid mesh is not constrained—applications can work with fine granular blocks. We study the fusion with a state-of-the-art shallow water solver at hands of an Intel Sandy Bridge and a Xeon Phi processor where we anticipate their reaction to selected block optimisation and vectorisation.

Keywords: Spacetrees, shallow water, adaptive Cartesian meshes, vectorisation, block fusion, shared memory parallelisation

1. Introduction

This paper addresses a conflict that many numerical simulations face. While the algorithms strive to reduce the number of unknowns and operations per accuracy via dynamic adaptivity in space and time, the hardware evolution asks for regular

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

data access patterns. Algorithms favour data structures and data access patterns that allow them to invest work where it pays off most. Recent hardware generations favour uniform sequential data access with high arithmetic intensity that allows to pipe data through the cores. The present paper investigates strategies to team up the advantages of adaptive, octree-type meshes with regularly refined patches (blocks). Plain shallow water equations act as test bed for our approach well-suited for numerous partial differential equations (PDEs). The first-mentioned model a wide range of problems of great societal and technical relevance: examples include tsunamis [12] or storm surges on the continental scale, radiation-sensitive cooling processes in manufacturing, as well as flow in blood vessels on the cell scale. Hyperbolic PDEs are often characterised by a multitude of scales in space and time, such that accurate solutions demand for very fine meshes in certain regions yet for a low time to solution, too. Tsunami prediction systems relying on hyperbolic simulations, e.g., have to yield results within minutes.

The multitude of scales of interest for hyperbolic solvers and their local yet transient behaviour in time imply that efficient computational meshes for these problems need to be dynamically adaptive. Furthermore, local time stepping is important where individual subgrids march in time with different time step sizes determined by the wave propagation speed. The finer the granularity of the adaptivity in both space and time, the "better" is the algorithm—at least in terms of the required number of unknowns and arithmetic operations.

If we express solvers with fine granular adaptivity in stencil notation, a large variety of computationally cheap stencils matching multiple local mesh refinement configurations is required. An application of a series of such stencils in turn exhibits non-uniform data access. However, modern multi- and manycore systems offering many hardware threads and broad vector facilities yield the best throughput for algorithms with low memory footprint and high arithmetic intensity that are split into a vast number of homogeneous tasks. This conflict of interest renders hyperbolic solvers on adaptive Cartesian grids a prototype challenge for novel high-performance computing architectures.

In the presented work, our meshes result from a k-spacetree formalism [16, 18] with $k = 2$ yielding a quadtree in two dimensions, where regular Cartesian grids—we denote them as *blocks*—are embedded into the leaves of the tree. Such a scheme facilitates dynamic, structured block adaptivity where the adaptivity leads to a low computational effort/memory footprint per accuracy ratio while a decent block size allows us to exploit vectorisation and loop parallelism. On the blocks, we apply the f-Wave wave propagation method to solve the Riemann problems with uniform vectorised stencils [1, 4, 12]. More sophisticated solvers fit to the scheme seamlessly. Yet, any increase in computational demands streamlines the challenge to design a high-throughput algorithm. The inter-block coupling is realised through bilinear conservative stencils in space and time from [14]. Similar techniques are proposed in [6, 7, 13], e.g., for other challenges.

If the size of the blocks can be chosen freely, multiple spacetrees induce the same

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

adaptive Cartesian grid—with a regular grid being a special case of an adaptive one. As a rule of thumb, big blocks induce high computational throughput. Small blocks in turn facilitate fine granular adaptive meshes. The latter gains importance if the application faces hard memory constraints or if local time stepping is realised on a per-block basis. In practice, one has to choose a block size compromise. In the present paper, we start from kernels of fixed size and study their computational potential with respect to vectorisation and shared memory parallelisation of their loops (*intra-block parallelism*). For an artificial test case problem, we also relate the block size selection to adaptive mesh refinement with local time stepping and a concurrent processing of multiple blocks by multiple threads (*inter-block parallelism*). Experiments show that the reduction in computational efficiency due to small block sizes is not always compensated by the reduction of total work due to adaptivity in space and time. It also becomes evident that the previously mentioned rule is invalid for big block sizes on some architectures and that the choice of one proper parallelisation variant or a hybrid depends on the block size and number of cores available. These insights do not answer which block size to select or what strategy to follow if some instationary regions of the grid require huge regular grids while others require very accurately trimmed adaptive meshes. We hence formalise the grid traversal as automaton running through the spacetree and augment this automaton with an analysed tree grammar [5]. Whenever the automaton encounters a set of spacetree leaves whose blocks can be fused into one bigger regular Cartesian block within the adaptive spacetree paradigm, these leaves are replaced accordingly—if the performance studies permit. This optimisation does not constrain the adaptivity pattern: once the grid refines in regions fused into a big regular grid, the automaton decomposes the block again.

The proposed technique falls into the class of autotuning of stencil codes for multicore SIMD architectures. It offers several selling points: The fusion of the blocks is hidden from the kernels just specified over block sizes. It does not increase the implementation complexity of the application code. The identification of grid regions well-suited to be fused is embedded into the tree traversal and anticipates dynamic adaptivity. It follows the grid evolution rather than opposing optimisation restrictions on the numerical algorithm's choice of discretisation. The loop fusion increases the *algorithmic homogeneity* of the data access pattern. It is independent of optimisations increasing the algorithmic intensity [8] though it can be combined with these. Finally, the optimisation can be tailored to distinct hardware characteristics without changing the compute kernels. Compared to our previous work [3], we detail the discussion with insights on the Xeon Phi architecture and particularities of the algorithms underlying the runtime tuning. New are an in-depth study of the two underlying parallelisation strategies with respect to hardware characteristics and the analysis of pessimistic vs. optimistic time stepping. We also start to put runtime improvements into relation to the mean life time of regular block assemblies.

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

Three research hypotheses drive the present paper:

(1) In terms of walltime, adaptivity with small block sizes as atomic mesh motif are not by nature superior to more regular grids. There are setups where a higher throughput of regular grids at least levels out a mesh cell increase.
(2) For these rather regular setups, the present approach exploits the studied architectures in terms of vector registers and cores. To study the methodology, getting as close to peak as possible is irrelevant. But the best throughput for the most advantageous block size acts as reference value and thus has to mirror the machine capabilities.
(3) Block fusion brings together the two advantages. It allows the application to select reasonably small block sizes while a high throughput is retained.

The remainder is organised as follows: We introduce our mesh formalism in combination with an abstract presentation of the unknown updates in Section 2 before we study the computational kernels in Section 3. The two orthogonal parallelisation schemes are sketched afterward. In Section 5, we introduce the block fusion. Some numerical results (Section 7) follow the description of our experiments in Section 6 and reveal the potential of the scheme. They also validate the underlying assumptions. A brief outlook and a summary in Section 8 close the discussion.

2. Shallow Water Equations on Spacetrees

Let $(0,1) \times (0,1) \subset \mathbb{R}^2$ be the bounding box of the computational domain. We cut this domain equidistantly into k parts along each coordinate axis. This yields k^2 non-overlapping cubes of the same size. If we continue this splitting recursively while we decide per cube autonomously whether to refine or not, we end up with an adaptive Cartesian grid. Let $\mathcal{C}$ be the set of all cubes resulting from the construction process. The refinement operation induces a parent-child relation $\sqsubseteq$ on $\mathcal{C}$ where each cube has either k^2 or no children at all. Cubes without children are *leaves* from the set $\mathcal{C}_L \subseteq \mathcal{C}$. The bounding box is the *root*.

The parent-child relation is a directed tree graph on $\mathcal{C}$ where a node's *level* is the path length from the root to the node. As the nodes of this graph are cubes, i.e. spatial elements, this tree is a *k-spacetree* [16]. $k = 2$ gives the special case of a quadtree. The *height* h of a spacetree is the length of the longest path in the graph. For the trivial spacetree with $\mathcal{C} = \mathcal{C}_L = \{(0,1) \times (0,1)\}$, we end up with height zero. All experiments of the present work are based upon the PDE framework Peano [17] and thus use $k = 3$. We hence omit the parameter k from now on and refer to that data structure variant as spacetree (Figure 1).

Volume-based discretisations of hyperbolic equations—or partial differential equations in general—such as finite volumes or finite elements directly yield stencils on any adaptive Cartesian grid induced by a spacetree formalism. While a direct spacetree-based stencil or system matrix derivation offers great flexibility with respect to the adaptivity, efficiency considerations as well as the intention to reuse

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

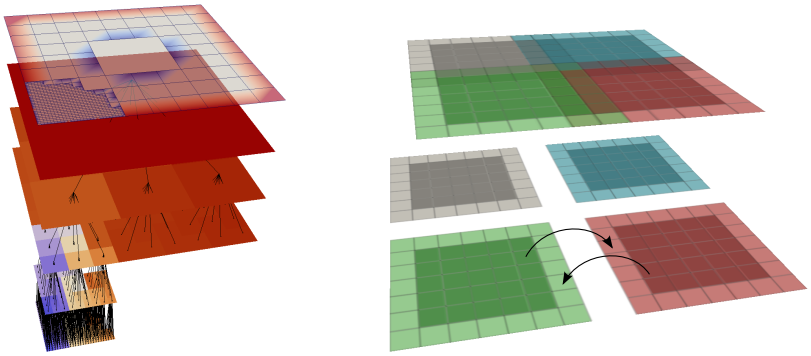

Fig. 1: Left: Adaptive Cartesian spacetree grid (top layer, transparent) with $k = 3$. The non-transparent layers below visualise the individual refinement steps, i.e. all elements of $\mathcal{C}$, with the tree relation $\sqsubseteq$ as black lines. Right: The grid (top) is decomposed into patches (below with $n = 5, \hat{n} = 1$). Ghost layer exchange for regular grids without local time stepping then is a plain copying of q^{old} into the neighbouring patches' ghost cells (cmp. arrows).

existing software fragments suggest to add an additional mapping $n : \mathcal{C}_L \mapsto \mathbb{N}$ that embeds an equidistant Cartesian mesh with $n(c) \times n(c)$ cells into each spacetree leaf. $n \equiv 1$ embeds a trivial grid of one cell into each leaf, i.e. each spacetree leaf is a cell of the computational grid Ω_h. In return,

$$n(c) = k^{\ell} \tag{1}$$

is equivalent to an $n \equiv 1$ spacetree grid created in two steps: we take a spacetree and embed an additional regular spacetree of height ℓ into each leaf.

In the present paper, we start from a fixed $n(c) = n \geq 2 \; \forall c \in \mathcal{C}_L$, and call the embedded regular Cartesian grids *blocks*. The spacetree then defines a block-structured adaptive Cartesian grid Ω_h, and it yields a non-overlapping domain decomposition of Ω_h. If we extend each $n \times n$ block by a halo layer of $\hat{n}$ cells, we obtain an overlapping domain decomposition. The unknowns per cell are denoted as q.

1: **for** $c \in \mathcal{C}_L$ **do**
2: Copy data from q^{old} of surrounding blocks into ghost cell entries of q^{old}
3: **for** $i \in \{\hat{n}, n + \hat{n} - 1\} \times \{\hat{n}, n + \hat{n} - 1\}$ **do**
4: $q_i^{new} \leftarrow computeNetUpdates(q^{old})$ ▷ Evaluates neighbours of cell i.
5: **end for** ▷ Loop also determines Δt.
6: **for** $i \in \{\hat{n}, n + \hat{n} - 1\} \times \{\hat{n}, n + \hat{n} - 1\}$ **do**
7: $q_i^{new} \leftarrow \Delta t \cdot q_i^{new} + q_i^{old}$ ▷ Time step update.
8: **end for**
9: **end for**
10: Switch q^{old} and q^{new} for all updated blocks

Given a stencil code mapping $(n + 2\hat{n}) \times (n + 2\hat{n})$ unknowns onto new values

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

within the $n \times n$ grid and intergrid operators mapping a $n \times n$ grid onto the halo layer of another grid, we can run over the spacetree's finest level, befill the halo layers of each individual block and update the unknowns (Algorithm **??**).

3. The Shallow Water Block Update Kernels

In the present paper, we solve the shallow water equations on $q = (\mathbf{h}, \mathbf{u}, \mathbf{v})$ given as

$$\partial_t \begin{bmatrix} \mathbf{h} \\ \mathbf{hu} \\ \mathbf{hv} \end{bmatrix} + \partial_x \begin{bmatrix} \mathbf{hu} \\ \mathbf{hu}^2 + \frac{1}{2} g \mathbf{h}^2 \\ \mathbf{huv} \end{bmatrix} + \partial_y \begin{bmatrix} \mathbf{hv} \\ \mathbf{huv} \\ \mathbf{hv}^2 + \frac{1}{2} g \mathbf{h}^2 \end{bmatrix} = S(t, x, y). \tag{2}$$

$\mathbf{h}$ denotes the height of the water column (water depth), $\mathbf{u}$ and $\mathbf{v}$ encode the momentum in x- and y-direction, and g is the gravitational constant ($g := 9.81\, m/s^2$). The source term $S(t, x, y)$ models effects of varying ocean depth (bathymetry) or frictional or Coriolis forces. In this paper, we neglect them. Our solver routines [2] realise an explicit finite volume scheme where a pair of unknown triples q is assigned to each cell of the grid to allow us to store the previous and the current time step. The six values are accompanied by a time stamp of the newer value plus the time interval spanned by the two solutions.

This leads to two computational kernels executed in each time step per block as soon as the halo layer also describing global boundary conditions is initialised:

- *Computation of net updates:* For each cell, we derive from the neighbouring cell quantities q^{old} the *net updates* $\Delta Q_h, \Delta Q_u, \Delta Q_v$. They determine the impact of waves on the cell quantities that enter or leave the respective grid cell through the edges. In the classical formulation, this step also derives from the wave speeds the biggest time step size Δt that one can chose without violating the CFL condition.
- *Updating the unknowns:* For each cell, the quantities in q then are then updated according to the balance equation

$$q^{new} = q^{old} - \frac{\Delta t}{k^h \cdot n} (\Delta Q_h, \Delta Q_u, \Delta Q_v). \tag{3}$$

The loop kernel to compute the net updates approximately solves a *Riemann problem* on each edge, i.e. solves the one-dimensional analogon of equation (2) for a piecewise constant initial condition given by the quantity vectors q_l and q_r obtained from the two adjacent cells. As approximate Riemann solver we follow a *wave propagation* approach by [4, 12], which determines the net updates from so-called *f-waves*, which are computed from a locally linearised Riemann problem. Our code provides a careful implementation of the resulting f-wave solver that allows auto-vectorisation [1]. The resulting net update kernel has a *computational intensity*, ratio of floating-point operations vs. accessed bytes of memory, of around two. In contrast, (3) has the ratio 2/3.

The ghost data exchange between different blocks is a copy of rectangular grid fragments as long as all blocks march in time with the same time step size and

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

induce the same mesh size. Each ghost cell then coincides with an inner cell of an adjacent block and q^{old} from there is copied into the ghost cells' q^{old}. If blocks can advance in time with different speed due to local time stepping, we have to interpolate linearly in time to determine the ghost layer's q^{old} from q^{old} and q^{new} from the adjacent block. For adaptive grids, we have to interpolate linearly both in time and space when we initialise the ghost layers, as the CFL condition makes the maximum time step size scale linearly with the mesh size. To facilitate this, we have to store per block the current and the previous time step. It is hence a natural choice to reuse half of these records to hold $\Delta Q = (\Delta Q_h, \Delta Q_u, \Delta Q_v)$ throughout the block update. Such an *in-situ* scheme allows us to write the block updates without additional temporary data structures. Technical details and a runtime model are given in [14].

Water height **h** and bathymetry usually allow us to predict the maximum time step size sharply. It is hence a natural choice to rewrite the *pessimistic* scheme from above determining the time step size from ΔQ into an *optimistic* variant: Here, we anticipate the scaling in (3) and merge the two algorithmic steps. If the net update computation afterward reveals that the CFL condition has been harmed, we can roll back the solution and restart the update with a halved time step size as q^{old} remains available. Roll-backs have never been observed for the present experiments.

4. Concurrency, Vectorisation and Parallelisation

The classic block-wise processing scheme exhibits two independent levels of concurrency. For the following discussion, we rely on a recursive formulation of the spacetree traversal (Algorithm **??**) where a push-back automaton traverses the tree mirroring a depth-first search and invokes the block updates on all leaves. Here, both the spacetree and unknown traversal exhibit concurrency.

```
1:  function ITERATE(c)
2:      parallel for c' ⊑ c do                                   ▷ Inter-block
3:          if c' ∈ C_L then
4:              Copy from q^old of surrounding blocks into ghost cells of q^old
5:              Determine Δt
6:              parallel for i ∈ {n̂, n+n̂−1} × {n̂, n+n̂−1} do      ▷ Intra-block
7:                  q_i^new ← q_i^old
8:              end parallel for
9:              parallel for i ∈ {n̂, n+n̂−1} × {n̂, n+n̂−1} do      ▷ Intra-block
10:                 q_i^new ← q_i^new + Δt · computeNetUpdates(q^old)
11:             end parallel for
12:             If necessary: rollback and restart computation with Δt/2
13:         else
14:             ITERATE(c')
15:         end if
16:     end parallel for
```

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

17: Switch q^{old} and q^{new} for all updated blocks
18: **end function**

We first discuss the downstream parallelism, i.e. the block updates. The two-dimensional loop iteration range of cardinality n^2 is free of write dependencies. Only when the loop terminates, we have to reduce the global time step size—either as input data for the subsequent algorithmic step or as rollback criterion. Given a maximum hardware concurrency level p, we can cut the kernel's image elements $q^{new}_{(\hat{n},\hat{n})}, q^{new}_{(\hat{n}+1,\hat{n})}, q^{new}_{(\hat{n}+2,\hat{n})}, \ldots, q^{new}_{(\hat{n}+n-1,\hat{n}+n-1)}$ either into segments of length p or p equally sized segments. Segments of length p induce SIMD *intra-block vectorisation*: one vector register befills p consecutive cells in the output array a time. p segments induce shared memory *intra-block parallelism*: one thread befills n^2/p consecutive cells in the image array parallel to the other threads.

Upstream, our traversal scheme exhibits concurrency on the block level. Children of one spacetree node can be processed concurrently for equidistant time stepping, as their kernel result depends only on q^{old}. For local time stepping or adaptive grids where interpolation in time is required, write races however occur. These are resolved by red-black colouring [5] or multiscale red-black colouring [15]. Given a maximum hardware concurrency level p, we can update up to p blocks descending from a refined node c in parallel. If we apply that scheme to SIMD, one vector register updates entries from p different blocks a time. Such an *inter-block vectorisation* equals a partial loop permutation of the inner and outer loop in Algorithm **??**. If applied to multiple threads, each of the p threads handles a distinct block before all threads continue with the subsequent p blocks. This is *inter-block parallelism*.

We can sophisticate the latter's block synchronous scheme by a task-based formalism where the dependencies between blocks stemming from local time stepping and adaptivity are represented by a graph. It is then up to the scheduler to resolve potential block update races [9]. Task-based parallelism here however is over-engineering. Our runtime per block depends linearly on the number of unknowns, i.e. the p block updates run equally long. This can change for more sophisticated Riemann solvers [1] where the update time per cell depends on entries in q^{old}.

5. Block Fusion

Obviously, the concurrency levels of the intra- and inter-block parallelisation depend on n. The bigger n the higher the intra-block concurrency. Big ns however make the underlying spacetree shallower for a given Ω_h. Consequently, the bigger n the smaller the inter-block concurrency. Once n is fixed, both concurrency levels are fixed for a given grid. We introduce a marker M on all spacetree nodes that characterises both the concurrency and the regularity of the grid locally:

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

$$M(c) = \begin{cases} 0 \text{ if } c \in \mathcal{C}_L \\ \hat{M} \text{ if } c \in \mathcal{C} \setminus \mathcal{C}_L \wedge \\ \quad \exists \hat{M} : \ \forall c_i \sqsubseteq c : (M(c_i) = m-1) \\ \bot \text{ else} \end{cases} . \tag{4}$$

We observe from (1) that we can take any node c in the spacetree with $M(c) > 0$ and replace all the blocks within the nodes deriving from c with a new block in c with $k^{M(c)}n \times k^{M(c)}n$ cells. Such a replacement preserves Ω_h.

Our block fusion algorithmic then reads as follows: First, traverse the spacetree with Algorithm **??**. Embed the computation of (4) into this traversal, i.e. compute the markers on-the-fly. Second, whenever the traversal encounters $M(c) > 0$ in subsequent traversals, embed a corresponding patch into this spacetree node c. Continue to traverse, but copy all $n \times n$ leaf blocks overlapping with the new fused block into the new data structure once the time step is completed. Flag these leaves afterwards and free their blocks. Third, whenever one encounters a flagged leaf, initialise the ghost data of the fused block on a coarser level instead of the ghost data associated to the leaf. Pointers to these data are inherited recursively throughout the traversal. Finally, whenever the call stack of the recursive traversal is reduced and it automaton ascends through a fused block, update this one. The fusion's interplay with dynamic adaptivity is obvious. If a spacetree leaf whose block got fused into a larger one identifies that the grid changes, it breaks down the fused block back into its $n \times n$ components. Such a break down is simple copying.

Block fusion enables us to shift the concurrency profile of a grid with small n from a high inter-block concurrency towards high intra-block concurrency on-the-fly. In practice, fusing wherever possible is not a good strategy. But it does make sense to establish a performance model predicting whether fusion along $M(c)$ levels pays off or it is better to fuse fewer levels along the spacetree hierarchy, and then to make use of the fusion paradigm. We also point out that any throughput improvement due to fusion in combination with a reduction of halo cell copying—within a fused block, $(k^{M(c)} - 1)^2$ fewer halo cell faces have to be exchanged per subsequent spacetree traversal, e.g.—first of all has to amortise a certain fuse overhead comprising the copying of block data into the fused block and back if the grid structure changes. Finally, we reiterate that fusion has an impact on local time stepping if the time stepping is realised per block. All three ingredients depend on solver, setup and hardware and thus fuse decisions should not be generic. The present paper hence highlights insights that can guide fusion. It does not study one particular fusion strategy.

6. Experiment Setup

The present experiments focus on Sandy Bridge and Xeon Phi processors instructed by the Intel compiler 14.0.2. Shared memory parallelisation is realised through Intel's TBB. The Sandy Bridge-EP Xeon E5-2650 processor with 2×8 cores and 4×16

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

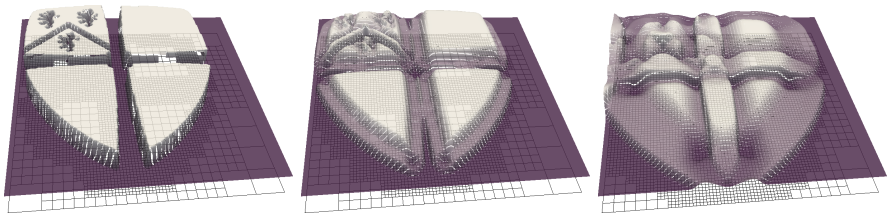

Fig. 2: The Durham University logo acts as sea level input for our shallow water equation solver.

GByte RAM at 2.0 GHz acts as driver for two Xeon Phi 5110P with 8GByte at 1.054 GHz. Experiments either run on the two 8-core processors or on one coprocessor in native mode. Hybrid runs, runs with two Phi coprocessors or runs on clusters of such setups are beyond scope. All figures are normalised runtimes (throughput) given as cell updates per second, i.e. computations of q^{new} divided by walltime. They include administrative cost such as maintaining the spacetree structure, determining (4) or fuse/inverse fusion copy overhead and are obtained in single precision.

Our testbed is an artificial wave propagation scenario starting from Figure 2 as initial water height and applies simple settings: The dynamic refinement criterion evaluates the maximum slope between the four corner points of each block and refines the corresponding leaf if this slope exceeds 0.01. It coarsens grid regions if the maximum slope of all contained blocks underruns 0.001. Our bathymetry is constant everywhere. We linearly scale the time step with the mesh size to meet the CFL condition as we set it to $0.001 \cdot 3^{-\ell}$. ℓ is the level of a block. All meshes are constrained by a minimum mesh size. If Ω_h is regular, this is the cell width. If Ω_h is adaptive, we start from a $3n \times 3n$ grid and make the refinement criterion refine constrained by the fact that no mesh cell may underrun the minimum mesh size.

All experimental setups rely on the Peano framework [17] and thus rely on three-partitioning. With $k = 3$, we start from $n \in \{6, 12\}$ as smallest block sizes. They are the smallest configurations where interpolation and restriction at grid resolution boundaries simplify as centers of coarse cells coincide with cell centers of finer and ghost meshes. For our search for advantageous block sizes in the huge n parameter space, we multiply these basic values with three mirroring k step by step.

7. Results

We first track the number of cell updates over time for different block sizes on adaptive grids (Figure 3). The smaller the block size n the smaller is the number of cell updates at a particular time as the total number of cells is the smaller the finer the adaptive granularity and as blocks with big cell sizes advance in time prior to neighbouring blocks with small cells. For the latter, we observe the expected 3:1 pattern: per time step of a coarse cell, the next finer cells have to perform three time steps. The impact of dynamic adaptivity is inrecognisable on a logarithmic

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

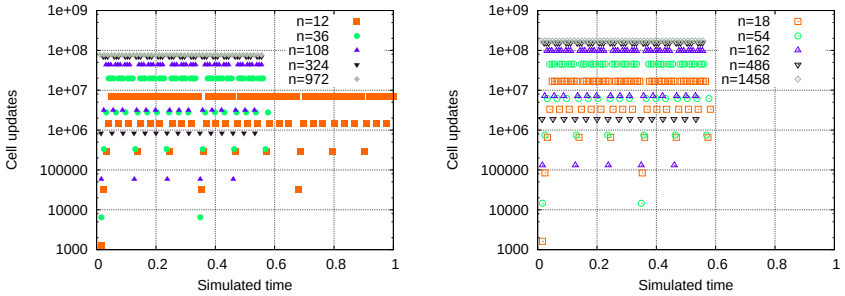

Fig. 3: Adaptive meshes with minimum mesh sizes 1/8748 (left) and 1/13122 (right).

scale for such a short observation interval.

We perceive that a reduction of the block size by a factor of nine ("two block sizes smaller") reduces the total number of cells around a factor of two. It reduces the memory footprint. The reduction pairs up with the fact that the few coarse cells have to be updated less frequently than fine blocks. Our patch update measurements reflect text book expectations and knowledge and do not contribute any new insight. However, the measurements characterise the interplay of block choice and workload footprint as well as workload homogeneity for the following experiments.

Observation. For the present experiments, a reduction of the block size reduces the number of arithmetic operations to run the simulation with a given accuracy.

We next study the arithmetic throughput on a single core. Hereby, we distinguish Xeon Phi from Sandy Bridge, optimistic from pessimistic time stepping and kernels vectorised with `#pragma simd` from kernels without annotation. For all measurements in this paper, the spacetree management required a single digit percentage of the overall runtime. We hence focus on the impact of manual SIMD-sation on the compute intensive block updates and the ghost cell exchange. Furthermore, only intra-block vectorisation made a performance difference whatever the setup. We ascribe this to the reduced bandwidth available to each vector register in this scheme and do not study it further though it might play a role on future architectures providing data gather and scatter such as AVX2.

For regular grids, we observe that the explicit usage of `simd` pragmas yields a speedup of around four on the Sandy Bridge as soon as the block size exceeds $n = 128$ (Table 1). For smaller blocks, the vectorisation is robust and increases linearly with the patch size. However, the loop ranges are too small to exploit all vector registers. Switching from pessimistic to an optimistic time stepping gives another five percent. From hereon, only optimistic time stepping is studied further. The measurements on the Xeon Phi reveal qualitatively the same behaviour (Table 2).

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

Table 1. Cell updates per second for regular mesh on Sandy Bridge.

patch size	pessimistic/simd	pessimistic/no-vec	optimistic/simd	optimistic/no-vec
6	$0.23 \cdot 10^7$	$0.22 \cdot 10^7$	$0.24 \cdot 10^7$	$0.23 \cdot 10^7$
12	$0.72 \cdot 10^7$	$0.51 \cdot 10^7$	$0.75 \cdot 10^7$	$0.53 \cdot 10^7$
18	$1.23 \cdot 10^7$	$0.71 \cdot 10^7$	$1.28 \cdot 10^7$	$0.73 \cdot 10^7$
36	$2.19 \cdot 10^7$	$0.95 \cdot 10^7$	$2.27 \cdot 10^7$	$0.97 \cdot 10^7$
54	$2.65 \cdot 10^7$	$1.02 \cdot 10^7$	$2.74 \cdot 10^7$	$1.01 \cdot 10^7$
108	$3.40 \cdot 10^7$	$1.05 \cdot 10^7$	$3.66 \cdot 10^7$	$1.08 \cdot 10^7$
162	$3.96 \cdot 10^7$	$1.08 \cdot 10^7$	$4.23 \cdot 10^7$	$1.10 \cdot 10^7$
324	$4.21 \cdot 10^7$	$1.12 \cdot 10^7$	$4.41 \cdot 10^7$	$1.13 \cdot 10^7$
486	$4.57 \cdot 10^7$	$1.13 \cdot 10^7$	$4.85 \cdot 10^7$	$1.14 \cdot 10^7$
972	$4.47 \cdot 10^7$	$1.13 \cdot 10^7$	$4.62 \cdot 10^7$	$1.14 \cdot 10^7$
1458	$4.43 \cdot 10^7$	$1.12 \cdot 10^7$	$4.77 \cdot 10^7$	$1.14 \cdot 10^7$
2916	$4.47 \cdot 10^7$	$1.11 \cdot 10^7$	$4.81 \cdot 10^7$	$1.13 \cdot 10^7$

The impact of the vectorisation however is—enabled by the hardware—twice as big. All results are in accordance with [1] working with higher clock rates.

Table 2. Cell updates per second for regular mesh on Xeon Phi.

patch size	pessimistic/simd	pessimistic/no-vec	optimistic/simd	optimistic/no-vec
6	$0.03 \cdot 10^7$	$0.03 \cdot 10^7$	$0.03 \cdot 10^7$	$0.03 \cdot 10^7$
12	$0.10 \cdot 10^7$	$0.06 \cdot 10^7$	$0.10 \cdot 10^7$	$0.06 \cdot 10^7$
18	$0.16 \cdot 10^7$	$0.08 \cdot 10^7$	$0.17 \cdot 10^7$	$0.08 \cdot 10^7$
36	$0.38 \cdot 10^7$	$0.11 \cdot 10^7$	$0.40 \cdot 10^7$	$0.11 \cdot 10^7$
54	$0.50 \cdot 10^7$	$0.11 \cdot 10^7$	$0.53 \cdot 10^7$	$0.12 \cdot 10^7$
108	$0.66 \cdot 10^7$	$0.12 \cdot 10^7$	$0.72 \cdot 10^7$	$0.12 \cdot 10^7$
162	$0.70 \cdot 10^7$	$0.12 \cdot 10^7$	$0.81 \cdot 10^7$	$0.12 \cdot 10^7$
324	$1.01 \cdot 10^7$	$0.13 \cdot 10^7$	$1.11 \cdot 10^7$	$0.13 \cdot 10^7$
486	$1.10 \cdot 10^7$	$0.13 \cdot 10^7$	$1.20 \cdot 10^7$	$0.13 \cdot 10^7$
972	$1.23 \cdot 10^7$	$0.13 \cdot 10^7$	$1.34 \cdot 10^7$	$0.13 \cdot 10^7$

Table 3. Cell updates for an adaptive mesh of minimum mesh size 1/26244.

patch size	max height	leaves	simd(sb)	no-vec(sb)	simd(phi)	no-vec(phi)
6	7	1170.24	$0.18 \cdot 10^7$	$0.18 \cdot 10^7$	$0.02 \cdot 10^7$	$0.02 \cdot 10^7$
12	7	1223.06	$0.54 \cdot 10^7$	$0.42 \cdot 10^7$	$0.07 \cdot 10^7$	$0.05 \cdot 10^7$
18	6	1053.52	$0.91 \cdot 10^7$	$0.59 \cdot 10^7$	$0.10 \cdot 10^7$	$0.06 \cdot 10^7$
36	6	1069.00	$1.49 \cdot 10^7$	$0.80 \cdot 10^7$	$0.16 \cdot 10^7$	$0.08 \cdot 10^7$
54	5	581.24	$1.83 \cdot 10^7$	$0.87 \cdot 10^7$	$0.21 \cdot 10^7$	$0.09 \cdot 10^7$
108	5	581.76	$2.30 \cdot 10^7$	$0.93 \cdot 10^7$	$0.23 \cdot 10^7$	$0.09 \cdot 10^7$
162	4	316.39	$2.94 \cdot 10^7$	$0.99 \cdot 10^7$	$0.33 \cdot 10^7$	$0.10 \cdot 10^7$
324	4	315.63	$3.03 \cdot 10^7$	$1.02 \cdot 10^7$	exceeds memory	
486	3	137.96	$3.90 \cdot 10^7$	$1.08 \cdot 10^7$	$0.58 \cdot 10^7$	$0.12 \cdot 10^7$
972	3	137.93	$3.74 \cdot 10^7$	$1.07 \cdot 10^7$		
1458	2	41.00	$4.77 \cdot 10^7$	$1.14 \cdot 10^7$		
2916	2	41.00	$4.80 \cdot 10^7$	$1.13 \cdot 10^7$		

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

We continue to rerun the experiments on adaptive grids where we constraint the finest mesh width to 1/26,244. Our measurements in Table 3 track the throughput, the maximum spacetree height and the number of blocks averaged over an observation interval of $t = 0.001$. Sandy Bridge preserves the regular grid throughput for reasonable big block sizes, for small n (36, e.g.) adaptive grids give around 70 percent of the regular throughput. Vectorisation yields a speedup of up to four. The adaptive grid even yields a slightly higher throughput than the regular case, maybe due to the tiling's positive impact on cache reuse [11]. Xeon Phi behaves differently. Adaptivity halves the throughput of the vectorised code while the variant without vectorisation is adaptivity invariant. Interpolation and restriction along resolution boundaries in this application field are predominantly memory movements, cannot be vectorised, and might cause cache misses though the measurements do not reveal the exact reason for this breakdown. From hereon, all experiments study the adaptive case.

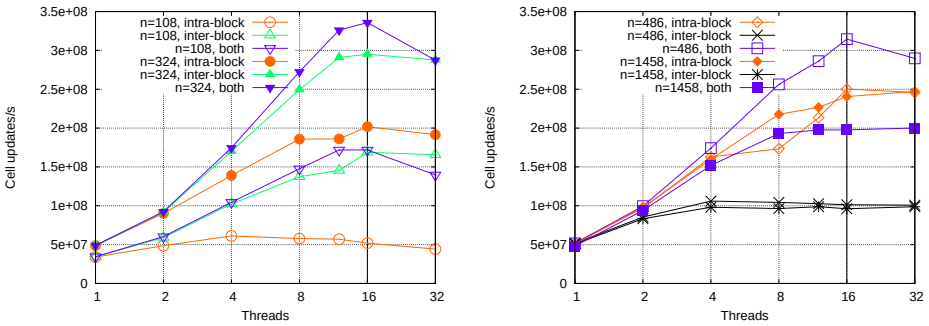

Fig. 4: Throughput for two different block sizes on Sandy Bridge with the two different shared memory parallelisation paradigms.

Observation. The present code can exploit vector instruction sets. On single cores, it is advantageous to make n as big as possible.

The simple "bigger patches-better throughput" paradigm becomes invalid once shared memory parallelisation is enabled. Selected runtimes are shown in Figures 4 and 5. On Sandy Bridge, we obtain the best throughput with $n = 324$. Up to this size, the impact of the inter-block parallelisation outweighs the intra-block benefit. Block sizes beyond 486 have a higher intra-block concurrency than an inter-block scheme. Starting from this size, the throughput becomes the worse the bigger n. Hyperthreading does not pay off. The Phis again behave differently. Two threads per core, i.e. half the physical thread count, here are the configuration of choice, while one out of 60 cores is reserved for the operating system. Furthermore, the throughput improves linearly with the block size. No deterioration threshold is observed. For reasonably big problem sizes, the coprocessor finally overtakes its host

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

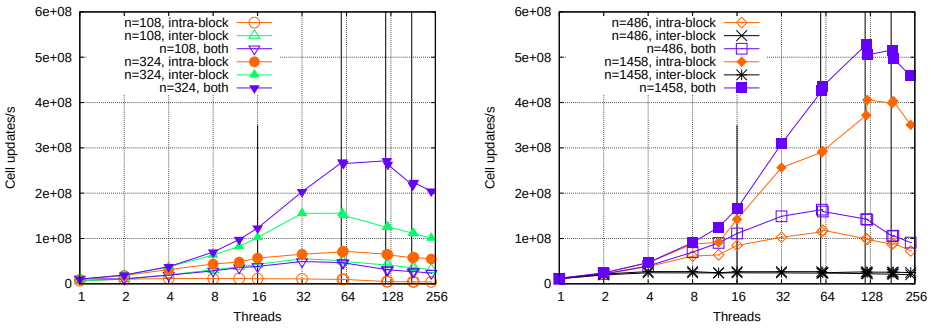

Fig. 5: Throughput for two different block sizes on Xeon Phi with the two different shared memory parallelisation paradigms. Additional vertical lines at `Threads` $\in \{59, 118, 177\}$.

mainly due to its ability to exploit the intra-block parallelism.

Observation. For ns saturating the scalability, we obtain an efficiency of around 7/16 on the Sandy Bridge and 10/60 on Xeon Phi. As the algorithm has low arithmetic intensity, we can state that it scales.

For real setups, plain throughput measurements are misleading. Instead, we have to put the throughput in relation to updates required, i.e. to science per flop [10]. In this case, the sweet spot moves to the left: it champions smaller block sizes. Although such a metric is strongly problem-dependent, we can derive examples from the present experiments. For a fixed scenario, we frequently observe a halving of computational load when we reduce the block size by one ninth. Whenever the throughput increases by more than a factor of two due to an increase of n by a factor of nine, it pays off to run for the bigger block size right away if memory permits. We observe such a behaviour on the Xeon Phi when the throughput of $\approx 5.2 \cdot 10^{‘8}$ for $n = 1458$ drops to a throughput smaller than $1.5 \cdot 10^8$ for $n = 162$ (not shown).

Observation. If scenarios cannot reduce the number of unknowns due to adaptivity significantly, it sometimes pays off to choose extensive large n to reduce the total time to solution.

We finally study block fusion. Previous experiments characterise the scalability and vectorisation suitability of certain block sizes. We know the potential gain of fusion. However, a quantification of the fusion overhead so far is missing: we do not know how expensive it is to move from one grid fragment into a fused representation, and how the fusion's fragmentation of the ghost layers affects the runtime. Transition cost means memory movements as the computation of (4) is computationally insignificant. These costs depend on the mean life time (mlt) of the blocks, i.e. the number of time steps before a fused blocks is destroyed again by the adaptivity criterion. While we switch off multicore parallelisation to study

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

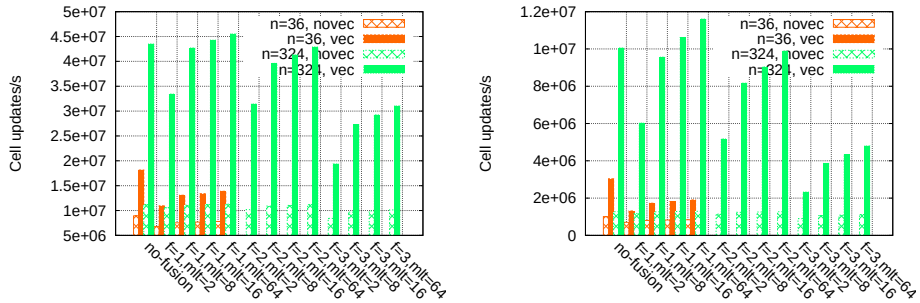

Fig. 6: Fusion on Sandy Bridge (left) and Xeon Phi (right) for different maximum fusion levels f and mean life time mlt.

the fusion overhead—otherwise overhead might be hidden behind other effects—vectorisation can have an impact as the vector architectures provide wide moves.

Our studies focus on one esemble of small blocks with $\hat{n} = 12$ in the grid. It is chosen such that the blocks can be fused into one block of size $n \in \{36, 324\}$ (Figure 6). Results for other configurations yield similar results. We start from the throughput for the n configuration (no-fusion) as best-case result and successively break it down into smaller blocks until we end up with the original block size $\hat{n} = 12$. For n, we obtain results in-between the span of regular to adaptive experiments. They are better than strongly adaptive runs as they average over the observation time and the grid becomes more regular. They are worse then regular runs as the grid is adaptive. For $mlt \geq 8$, we observe that the block transition cost is amortised, i.e. the fused throughput becomes saturated. For $l = 1$, i.e. the fusion of 3×3 blocks, our results recover the throughput of the big block sizes. A similar reasoning holds for the fusion of 27×27 blocks. If we fuse bigger esembles, the fusion stagnates with two third of the best-case throughput on Sandy Bridge. On the Xeon Phi, we obtain just half of the throughput.

Observation. Due to on-the-fly block fusion we can preserve the throughput of block sizes that are up to nine times bigger than the chosen size.

8. Conclusion and Outlook

The present paper studies a dynamically adaptive shallow water equation solver that starts from an adaptive spacetree and embeds blocks of fixed size into the spacetree's leaf nodes. Our results confirm the natural intuition that the throughput of the solver is, as rule of thumb, the better the bigger the block sizes and they show that the two introduced parallelisation concepts have the potential to exploit current hardware. As small block sizes are desirable due to memory and total time-to-solution considerations, we propose an on-the-fly identification of regular grid regions and fuse the blocks there into big data chunks. A study on the overhead of

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

this loop fusion reveals that dynamic block fusion helps to harvest scalability and vectorisation characteristics of big blocks sizes though the algorithm may work with small blocks of small memory footprint that track the solution characteristics accurately. However, this statement holds if there are reasonably structured, i.e. regular, grid regions and the mean time to reconstruction is not excessively small.

The proposed performance studies rely on one simple partial differential equation solver. However, we consider our algorithmic principles to be of potential relevance for a broader community. When the techniques are adopted, it is however of relevance to switch from manually tuned kernels to automatically tuned loop assemblies as generated by modern stencil compiler, i.e. to benefit from the combination of the present approach improving the arithmetic data access homogeneity with techniques increasing the arithmetic intensity [8]. One enabling code feature for this future work—ghost layers with $\hat{n} \geq 2$—is sketched.

Acknowledgements

Tobias Weinzierl appreciates the support of the School of Engineering and Computing Sciences and in particular Tomasz Koziara at Durham University for providing the Xeon Phis. Michael Bader appreciates the support of the SFB/TRR 89 (Transregional Collaborative Research Centre) *Invasive Computing* funded by the German Research Foundation (DFG). Kristof Unterweger and Roland Wittmann appreciate the support of Award No. UK-c0020, made by the King Abdullah University of Science and Technology (KAUST). All software is freely available at [2, 17].

References

[1] M. Bader, A. Breuer, W. Hölzl, and S. Rettenberger. Vectorization of an augmented riemann solver for the shallow water equations. In *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*, pages 193–201, 2014.

[2] M. Bader, A. Breuer, and S. Rettenberger. SWE—the Shallow Water Equations teaching code, 2013. https://github.com/TUM-I5/SWE.

[3] M. Bader, A. Breuer, S. Rettenberger, K. Unterweger, T. Weinzierl, and R. Wittmann. Hardware-aware block size tailoring on adaptive spacetree grids for shallow water waves. In A. Größlinger and H. Köstler, editors, *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, pages 57–64, 2014.

[4] D.S. Bale, R.J. LeVeque, S. Mitran, and J.A. Rossmanith. A wave propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM Journal on Scientific Computing*, 24(3):955–978, 2003.

[5] W. Eckhardt and T. Weinzierl. A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, PPAM 2009*, volume 6068 of *LNCS*, pages 567–575. Springer-Verlag, 2010.

[6] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. WaLBerla: HPC software design for computational engineering simulations. *Journal of Computational Science*, 2(2):105–112, 2011.

Preprint of an article published in Parallel Processing Letters, 24, 3, 2014
10.1142/S0129626414410060 ©World Scientific Publishing Company

[7] J. Frisch, R.-P. Mundani, and E. Rank. Adaptive distributed data structure management for parallel CFD applications. In *Proc. of the 15th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2013. accepted.

[8] P. Ghysels and W. Vanroose. Modeling the performance of geometric multigrid on many-core computer architectures. *SISC*, 2014. submitted.

[9] P. Gonnet. Quicksched: Task-based parallelism with dependencies and conflicts. Technical Report ECS-TR 2013/06, School of Engineering and Computing Sciences, Durham University, South Road, DH1 3LE Durham, United Kingdom, 2013.

[10] D. E. Keyes. Four Horizons for Enhancing the Performance of Parallel Simulations Based on Partial Differential Equations. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2000.

[11] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies 2002*, pages 213–232. Springer, 2003.

[12] R. J. LeVeque, D. L. George, and M. J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, 2011.

[13] P. Neumann. *Hybrid Multiscale Simulation Approaches For Micro- and Nanoflows*. Verlag Dr. Hut, München, 2013.

[14] K. Unterweger, T. Weinzierl, D. Ketcheson, and A. Ahmadia. Peanoclaw—a functionally-decomposed approach to adaptive mesh refinement with local time stepping for hyperbolic conservation law solvers. Technical report, TUM, 2013.

[15] M. Weinzierl. *Hybrid Geometric-Algebraic Matrix-Free Multigrid on Spacetrees*. Dissertation, Fakultät für Informatik, Technische Universität München, München, 2013.

[16] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Verlag Dr. Hut, 2009.

[17] T. Weinzierl et al. Peano—a Framework for PDE Solvers on Spacetree Grids, 2012. www.peano-framework.org.

[18] T. Weinzierl and M. Mehl. Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, October 2011.